Traducción de proyectos con GNU gettext en 15 minutos

José Tomás Tocino García

Hackathon UCA - Diciembre 2010

Índice

1. Introducción

GNU gettext es un conjunto de herramientas libres de internacionalización que permite traducir nuestros proyectos de una manera sencilla. Es el sistema de i18n [1] más utilizado, y lo podéis encontrar en una gran cantidad de proyectos.

Las ventajas de usar un sistema como *gettext* en lugar del típico menú de elección de idiomas dentro de la aplicación son muchas. Por ejemplo, con gettext el ajuste del idioma es transparente al usuario. Además, ofrece todas las ventajas de usar un sistema establecido y prácticamente estándar.

[1] *"i18n"* es una alternativa a escribir *internacionalización*, el 18 simboliza los 18 caracteres entre la *i* inicial y la *n* final

1.1. Internacionalización vs localización

La *internacionalización* de un proyecto consiste en prepararlo de forma que sea capaz de trabajar y presentarse en una multitud de idiomas. Por otro lado, la *localización* trata de coger un programa internacionalizado y darle la suficiente información para que se adapte al idioma y configuración del usuario actual.

1.2. El paquete de herramientas de GNU gettext

GNU gettext está compuesto de un conjunto de elementos que forman un *framework* de trabajo común. En particular, esto incluye:

- Una serie de directivas y convenciones a la hora de escribir un programa.
- Un esquema estándar de organización de ficheros y directorios para guardar los ficheros relacionados con la traducción (¿Os suena `LC_MESSAGES`?)
- Una biblioteca que provee funciones relacionadas con las cadenas traducidas.
- Algunas utilidades para la creación y edición de los ficheros con las cadenas de traducción.
- Un modo para **Emacs** :)

2. Pasos en el proceso de traducción

A la hora de adaptar un proyecto para internacionalizarlo hay que seguir una serie de pasos bastante mecánicos que se repiten cada vez que añadamos o modifiquemos las cadenas de nuestro proyecto.

2.1. Adaptación del código fuente

Primero, necesitamos adaptar el código de nuestro programa, marcando de alguna forma las cadenas que queremos que queden traducidas. Podría pensarse que este paso podría ser automático, pero hay cadenas que a buen seguro no queremos que sean traducidas, como por ejemplo parámetros de funciones o cosas así.

Para ello, añadiremos las siguientes líneas al inicio de nuestro fichero fuente:

```
1 #include <libintl.h>
2 #include <locale.h>
3 #define _(x) gettext(x)
```

La biblioteca `libintl` será la encargada de la internacionalización de nuestro proyecto, siendo de especial interés su función `char * gettext (const char *)`, a la que le pasaremos las cadenas originales y nos devolverá la cadena traducida dependiendo del *locale* del sistema. Para hacer menos evidente su uso, utilizamos la macro definida arriba.

> **Nota:** Para ahorrarnos trabajo y problemas, el idioma *por defecto* será el inglés, ya que los ficheros de traducción iniciales se generan en inglés y *no he encontrado* ninguna forma de cambiarlo.

Lo siguiente será, lógicamente, cambiar todas las cadenas que queramos traducir en nuestro proyecto, de forma que en realidad sean siempre llamadas a la función `gettext`, o preferiblemente a su macro `_()`.

```
  string mensaje = "Hello world";
```

Pasará a ser:

```
  string mensaje = _("Hello world");
```

Algo a tener en cuenta es que es necesario elegir un nombre para el *dominio* de la internacionalización, que por regla general coincidirá con el proyecto. En nuestro ejemplo, este nombre será **jakatoner** (del inglés *hackathon* y el sufijo ibérico *-er*).

En las primeras líneas de nuestro proyecto añadiremos las siguientes instrucciones de inicialización:

```
bind_textdomain_codeset ("jakatoner", "UTF-8");
setlocale(LC_ALL, "");
bindtextdomain("jakatoner", "lang" );
textdomain("jakatoner");
```

Eso le indicará a la biblioteca de i18n cuál es el dominio de la traducción, así como el directorio de las traducciones (`lang`), y pondrá el locale por defecto.

Así pues, un posible fichero `main.cpp` de ejemplo podría ser:

```
#include <iostream>
#include <libintl.h>
#include <locale.h>

#define _(x) gettext(x)

using namespace std;

int main(int argc, char *argv[])
{
    bind_textdomain_codeset ("jakatoner", "UTF-8");
    setlocale(LC_ALL, "");
    bindtextdomain("jakatoner", "lang" );
    textdomain("jakatoner");

    cout << _("Hello world") << endl;
    return 0;
}
```

2.2. Generando los ficheros de traducción

Una vez que tengamos todas las cadenas a traducir apropiadamente adaptadas como se ha comentado antes, pasaremos a crear los ficheros con los que realizaremos las traducciones. Hay varios tipos de estos ficheros:

.POT *(Portable Object Template)* Es el primer fichero que se genera, y contiene todas las cadenas extraídas del código fuente, que servirá luego como plantilla para los ficheros `.po`.

.PO *(Portable Object)* Son los ficheros principales de traducción, los que se editan con las cadenas traducidas. Hay uno por cada *locale* que queramos incluir.

.MO *(Machine Object)* Son la versión binaria de los ficheros `.po`, los que nuestra aplicación usará para leer las cadenas traducidas.

La forma de organizar los diferentes ficheros está bastante estandarizada, de forma que lo más recomendado es seguirla – de no hacerlo podemos tener problemas a la hora de que el programa encuentre los ficheros de traducción.

- En la raíz de nuestro proyecto tendremos una carpeta **po** que albergará el fichero de plantilla `.pot`, en nuestro caso `jakatoner.pot`, así como los ficheros `.po` de cada *locale*: `en.po`, `es.po`, etc.
- Además, también en la raíz tendremos otra carpeta llamada **lang**. Dentro de ella habrá una carpeta por cada fichero `.po` en la carpeta antes mencionada, y dentro de cada una de ellas, una carpeta `LC_MESSAGES`, que albergará el fichero `.mo` correspondiente, todos con el nombre del dominio, en nuestro caso `jakatoner.mo`

La estructura de directorios que obtendremos será algo así:

```
lang
|-- en
|   ‘-- LC_MESSAGES
|       ‘-- jakatoner.mo
‘-- es
    ‘-- LC_MESSAGES
        ‘-- jakatoner.mo
po
|-- en.po
|-- es.po
‘-- jakatoner.pot
```

2.2.1. Creando la plantilla .pot

Así pues, el primer paso será generar el fichero `.pot`. Para ello utilizaremos la utilidad `xgettext`, que tiene *una pechá* de opciones. Así pues, creamos los directorios antes comentados y usamos la siguiente expresión:

```
xgettext --package-name jakatoner --package-version 0.1 \
--default-domain jakatoner --output po/jakatoner.pot --from-code=utf-8 \
--copyright-holder="Tu nombre" --msgid-bugs-address="tu@mail.com" \
-s -k_ -C main.cpp
```

La mayoría de las opciones son autoexplicativas, pero es interesante conocer el significado de las que no lo son:

-s Salida ordenada, ordena las cadenas en el fichero de plantilla, útil cuando tenemos muchos ficheros fuente y queremos tener las cadenas organizadas.

-k_ Indica que también busque cadenas marcadas con `_(cadena)` además de `gettext(cadena)`.

-C Indica que el lenguaje es C++.

Con esto tendremos el fichero en `po/jakatoner.pot`. Es necesario editarlo y cambiar el valor de `CHARSET` (en la línea `Content-Type: ...`) por `UTF-8`, `xgettext` aún no ofrece ninguna opción para autorrellenar este campo.

2.2.2. Creando los ficheros de traducción .po

El siguiente paso será el de crear un fichero `.po` para cada uno de los idiomas a los que queramos traducir nuestro proyecto. Para ello utilizaremos la utilidad `msginit` de la siguiente manera, suponiendo los idiomas inglés y español:

```
msginit -l es -o po/es.po -i po/jakatoner.pot
msginit -l en -o po/en.po -i po/jakatoner.pot
```

Al ejecutar los comandos nos pedirán nuestro email en un mensaje bastante utópico en el que describe un mundo en el que el feedback del usuario realmente existe más allá del *"¡¡Tu programa es una mierda!!"*

Si le echamos un vistazo al fichero `po/es.po`, aparte de toda la morralla inicial, nos encontraremos con las cadenas de nuestro programa de la siguiente manera:

```
#: main.cpp:15
msgid "Hello world"
msgstr ""
```

El formato es muy sencillo: la primera línea indica la situación de la cadena en el código fuente, la segunda es la cadena original, y la tercera es la cadena traducida. Para el fichero en español, quedaría:

```
#: main.cpp:15
msgid "Hello world"
msgstr "Hola mundo"
```

Cabe notar que como el lenguaje original es el inglés, podemos dejar el fichero `en.po` intacto, ya que por defecto utilizará las cadenas originales.

2.2.3. Creando los binarios .mo

Una vez terminado el apartado anterior, nos queda el último paso, que es generar los ficheros binarios con las traducciones. Para ello, utilizaremos la utilidad `msgfmt`, que convertirá los `.po` en `.mo`, de la siguiente manera:

```
mkdir lang/{es,en}/LC_MESSAGES
msgfmt -c -v -o lang/es/LC_MESSAGES/jakatoner.mo po/es.po
msgfmt -c -v -o lang/en/LC_MESSAGES/jakatoner.mo po/en.po
```

La opción **-c** indica que se hagan chequeos ante errores, y la opción **-v** muestra una salida extendida (*verbose*). Con esto, ya tendremos todos los ficheros necesarios.

2.3. Compilación y ejecución

Para compilar nuestro proyecto, si utlizamos GCC no será necesario enlazar a ninguna librería especial, puesto que `libintl` ya viene en la biblioteca estándar. Así pues, *good old gcc, here we go*:

```
g++ -o jakatoner main.cpp
```

Tras esto, podremos probar nuestro programa:

```
jose@jose-desktop:~$ ./jakatoner
Hola mundo
jose@jose-desktop:~$ LANG=en_UK ./jakatoner
Hello world
```

2.4. Mantenimiento

Supongamos ahora que añadimos una línea más a nuestro código, en la que se utiliza una cadena nueva. Si seguimos el proceso anterior perderemos todas las traducciones que ya teníamos, ya que los ficheros se crearían de cero. Para evitar esto, GNU gettext ofrece una utilidad llamada `msgmerge` que nos permitirá actualizar los ficheros `.po` con las nuevas cadenas manteniendo las traducciones ya realizadas.

Así pues, supongamos que añadimos esta línea al fichero:

```
cout << _("Bye bye, dear user") << endl;
```

Generamos el fichero de plantilla `.pot` igual que lo hicimos antes, pero a la hora de generar los ficheros `.po` utilizaremos lo siguiente:

```
xgettext --package-name jakatoner --package-version 0.1 \
--default-domain jakatoner --output po/jakatoner.pot --from-code=utf-8 \
--copyright-holder="Tu nombre" --msgid-bugs-address="tu@mail.com" \
-s -k_ -C main.cpp

msgmerge -s -U po/es.po po/jakatoner.pot
msgmerge -s -U po/en.po po/jakatoner.pot
```

La opción **-s** genera una salida ordenada, y **-U** indica que la operación es de actualización (*update*). Con esto, ya tendremos el fichero `.po` con las nuevas cadenas añadidas y las cadenas antiguas sin cambios, podremos proceder a añadir las traducciones y generar los ficheros `.mo` tal y como se ha explicado previamente.

```
msgfmt -c -v -o lang/es/LC_MESSAGES/jakatoner.mo po/es.po
msgfmt -c -v -o lang/en/LC_MESSAGES/jakatoner.mo po/en.po

./jakatoner
Hola mundo
Nos vemos, querido usuario

LANG=en_UK ./jakatoner
Hello world
Bye bye, dear user
```

3. Goodies finales

3.1. PO-mode en Emacs

Como comenté al principio, existe un modo de Emacs para la edición eficiente de archivos `.po`. Puede instalarse manualmente de la manera habitual o en sistemas basados en paquetería deb con el siguiente comando:

```
sudo apt-get install gettext-el
```

Una vez hecho esto, al abrir un fichero `.po` entraremos en el modo PO (podemos forzarlo con `M-x po-mode`). Hay gran cantidad de comandos para editar los ficheros PO, pero los más útiles son los siguientes:

- Con **n** y **p** iremos al siguiente o anterior mensaje de traducción.
- Para saltar entre los mensajes traducidos usaremos **t** y **T**. Para los no traducidos, **u** y **U**.
- Para editar la traducción, pulsamos **Intro**, que abrirá un marco con el mensaje a editar. Tras modificarlo, podemos confirmar los cambios con **C-c C-c** o cancelarlos con **C-c C-k**
- Podemos acceder a la ayuda en cualquier momento pulsando **h**.

Una vez acostumbrados, la edición de estos ficheros se hará mucho más liviana y rápida. Existen, de cualquier modo, editores íntegramente dedicados a la edición de ficheros .po.

3.2. Referencia

Para ampliar conocimientos sobre GNU gettext, lo mejor es dirigirse a la referencia oficial [2] que, aunque bastante extensa, resulta muy interesante y amena de leer, explicando toda clase de casos especiales de traducción, como aquellos en los que aparecen cadenas de formato relacionadas con sentencias al estilo de `printf` y otros casos particulares.

[2] http://www.gnu.org/software/gettext/manual/gettext.html